REMARKS

This Amendment is in response to the Office Action dated May 5, 2004. In the Office Action, the Examiner rejected claims 1-30 under 35 U.S.C. § 103(a) as being unpatentable over Datta, U.S. Patent No. 5,671,422 (hereinafter *Datta*), in view of Kimura *et al.*, U.S. Patent No. 5,737,579 (hereinafter *Kimura*).

Claims 1, 3, 4, 5, 10, 23-25, and 28-30 are amended as shown above. In particular, independent claims 1, 23, and 28 are amended to more clearly recite features of the claimed invention. Claims 1-30 remain pending in the application. For the reasons set forth below, the Applicants respectfully request reconsideration and allowance of all pending claims.

CLAIM REJECTIONS - 35 U.S.C. § 103

To establish a *prima facie* case of obviousness, there must first be some suggestion or motivation to modify a reference or to combine references, and second be a reasonable expectation of success. The teaching or suggestion to make the claimed combination and the reasonable expectation of success must both be found in the prior art and not based on applicant's disclosure. Third, the prior art reference (or references when combined) must teach or suggest all the claim limitations. M.P.E.P. § 706.02(j) from *In Re Vaeck*, 947 F.2d 488, 20 USPQ2d 1438 (Fed. Cir. 1991). Where claimed subject matter has been rejected as obvious in view of a combination of prior art references, a proper analysis under § 103 requires, *inter alia*, consideration of two factors: (1) whether the prior art would have suggested to those of ordinary skill in the art that they should make the claimed device; and (2) whether the prior art would also have revealed that in so making, those of ordinary skill would have a reasonable expectation of success. Both the suggestion and the reasonable expectation of success must be founded in the prior art, not in the Applicants' disclosure. *Amgen v. Chugai Pharmaceutical*, 927 F.2d 1200, 18 USPQ2d 1016 (Fed. Cir. 1991), *Fritsch v. Lin*, 21 USPQ2d 1731 (Bd. Pat. App. & Int'f 1991). An invention is non-obvious if the references fail not only to expressly disclose the claimed invention as a whole, but also

to suggest to one of ordinary skill in the art modifications needed to meet all the claim limitations. *Litton Industrial Products, Inc. v. Solid State Systems Corp.*, 755 F.2d 158, 164, 225 USPQ 34, 38 (Fed. Cir. 1985).

The examiner must present a convincing line of reasoning as to why the artisan would have found the claimed invention to have been obvious in light of the teachings of the references. M.P.E.P. § 70602(j) from *Ex parte Clapp*, 227 USPQ 972, 973 (Bd. Pat. App. & Inter. 1985). Obviousness cannot be established by combining references without also providing evidence of the motivating force which would impel one skilled in the art to do what the patent applicant has done. M.P.E.P. § 2144 from *Ex parte Levengood*, 28 USPQ2d 1300, 1302 (Bd. Pat. App. & Inter. 1993) (emphasis added by M.P.E.P.).

In support of the claim rejections, the Examiner specifically considered the elements of claims 28-30, while asserting that claim 1-22 and claims 23-27 recited analogous elements. (This assertion is partly correct, and partly not, as discussed below). In support of the rejection of independent claim 28, the Examiner states,

> As to claim 28, Datta teaches a motherboard on which an original set of firmware is stored, a memory operatively coupled to the motherboard in which a plurality of machine instructions are stored, and a processor linked in communication with the memory for executing the machine instructions (figure 1, col. 4, lines 25-48). Datta also teaches performing the operations of providing a mechanism to enable an event handler that is accessible to a hidden execution and storage mode of the processor bit is not accessible to other operating modes of the processor, and executing the event handler in response to an event that causes the processor to be switched to the hidden execution and storage mode to service the event (abstract and col. 2 lines 1-23).

The Examiner then acknowledges that *Datta* "does not expressly teach executing the machine instructions to enable a loading of an event handler that is not stored in an original set of firmware as supplied by an original equipment manufacturer (OEM) of the computer system into the hidden memory space." In view of this shortcoming, the Examiner asserts that such elements are taught by *Kimura*, with specific reference to col. 69 lines 30-39. The Examiner then states,

> It would have been obvious for one or ordinary skill in the art at the time of the invention to combine the teachings of Datta and Kimura et al because both are commonly directed to the SMM handler environment and Kimura et al's loading option of external SMM handler, when incorporated into Datta's system, would have enabled increased architectural flexibility by allowing a configuration with an outside bus arbiter to perform processing mode switching.

Claim 28 has been amended herein, and now recites.

28. A computer system comprising:

a motherboard on which an original set of firmware is stored;

a memory operatively coupled to the motherboard in which a plurality of machine instructions are stored; and

a processor linked in communication with the memory, for executing the machine instructions to perform the operations of:

loading *a first event handler that is stored in an original set of firmware as supplied by an original equipment manufacture (OEM) of the computer system* into a hidden memory space that is accessible to the hidden execution and storage mode but is not accessible to other operating modes of the processor;

loading *a second event handler that is not stored in the original set of firmware as supplied by an original equipment manufacture (OEM) of the computer system* into a the hidden memory space; and

enabling selective execution of the first and second event handlers in response to an event that causes the processor to be switched to the hidden execution and storage mode to service the event. (Emphasis added)

Applicant respectfully asserts that amended claim 1 is patentable over the cited art.

As discussed above, *Datta* "does not expressly teach executing the machine instructions to enable a loading of an event handler that is not stored in an original set of firmware as supplied by an original equipment manufacturer (OEM) of the computer

system into the hidden memory space." Rather, this element is asserted to be taught by *Kimura*.

While *Kimura* provides a scheme that enables execution of an event handler that is not included as part of the original set of firmware as supplied by an original equipment manufacture (OEM) of a computer system, *Kimura*'s system and method do not also support loading a first event handler that *is* stored in an original set of firmware as supplied by an original equipment manufacture (OEM) of the computer system into a hidden memory space and loading a second event handler that *is not* part of the OEM firmware, and enable selective execution of the first and second event handlers as recited in amended claim 28. The following examination of how *Kimura*'s method and system works reveals why this is not possible.

Kimura is directed to a system and method for emulating computer architectures. In particular, *Kimura* employs the system management mode (SMM) of an Intel Pentium® processor to facilitate the emulation technique. In doing so, *Kimura* redirects a pointer for the location of an SMM handler from a default location to a selected address at which the SMM handler resides. In one embodiment, the selected address is the starting address of a "rear" portion of VRAM (video RAM) (see, *e.g.*, Col. 34, lines 27 through Col. 35, line 12). *Kimura* also indicates that other memory resources may be used for SMRAM (system management random access memory) purposes. This consideration is presumably supported of the statements made in Col. 69, lines 30-39 (as referenced above) of:

> With a configuration in which components such as the subcontroller are located on an option board, programs such as the SMM handler may be stored in an external storage device such as a floppy disk or CD-ROM. This software may be loaded into RAM of a personal computer system to enable the SMM handler to execute the given processing. The RAM of a personal computer system of the first architecture may be converted to act as SMRAM, or an independent storage device may be provided on a board to use as SMRAM.

As discussed in Col. 34, lines 27 through Col. 35, line 12, the default SMBASE value of 38000h (technically, the default is 30000h plus an offset of 8000h - see the

attached material discussing System Management Mode for Intel processors) is changed such that SMM (the SMM handler) is now started in VRAM. A similar technique could be used to start the SMM handler in a selected portion of RAM. The actual starting address is stored in a processor register, and thus this register value is modified when SMBASE is redirected to the beginning of the rear portion of VRAM. While the System Management Mode architecture provides a means to support this functionality, it defeats the conventional or "legacy" usage of SMM, as follows.

SMM was developed to handle certain processing operations in a manner that is transparent and apart from an operating system running on the computer system. Thus, the operating system is unaware of when the processing mode switches to SMM. Furthermore, the operating system should be unaware of any SMM resources, SMRAM in particular.

Under the conventional approach, a portion of system RAM (*e.g.*, a 64-Kbyte region) is reserved from SMRAM purposes by the system BIOS during system initialization. When the RAM address map is handed off to the operating system during OS boot, the region of system RAM reserved for SMRAM purposes is not revealed to exist to the operating system, and thus the operating system is unaware this portion of system RAM exists.

Under the *Kimura* scheme that employs the rear portion of VRAM, the operating system is also unaware of this usage, since VRAM is typically used for storing video data, and is not used as a system memory resource. This scheme could be extended to a memory resource that is on an option card, since, again, the operating system will typically by unaware of this type of memory resource, unless specifically made aware of it by the system BIOS during OS boot.

Returning to the conventional approach, the system BIOS is used to setup and manage SMRAM. Thus, the system BIOS needs to be aware of the location of SMRAM. More importantly, legacy firmware code that is employed to handle SMM events is coded so that it operates at a specific location in memory. As shown on page

13-4 of the Intel system management reference, the conventional location of SMRAM is from [SMBASE +8000H] to [SMBASE + FFFFH]. Thus, the legacy firmware code is expected to be located in this address range.

As discussed in the Background of the Invention section of the present application, this SMM handler code comprises 16-bit assembly in IA32 (32-bit Intel Architecture) processors, such as Pentium processors, and 64-bit assembly for Itanium processors. Under the conventional approach, the SMM handler code is a monolithic block of firmware that is loaded into the default SMM address range during system initialization. In order for the 16-bit assembly code to properly work, the BIOS and SMM handler needs to be aware of where it is located.

Under *Kimura*, the add-on SMM handler (that is, the handler code that is not part of the SMM handler code supplied by the system OEM) is also a monolithic block of software code that is located at some remapped location, such as the rear portion of VRAM, system RAM, or a memory resource of an option board. In response to an SMI event, normal SMI operations are performed (*e.g.*, stack store, *etc.*), as discussed under the section "Description of SMI Processing", beginning at Col. 21, line 61. The instruction pointer for the processor is then redirected to the new SMBASE location. Thus, the processing is redirected to the location of the add-on SMM handler. This means there is no way to access the OEM SMM handler code, as there are no provisions in the add-on SMM handler that identify the location of this OEM SMM handler code. Thus, there is no way that *Kimura* can support "enabling selective execution of the first (the OEM) and second (the non-OEM) event handlers in response to an event that causes the processor to be switched to the hidden execution and storage mode to service the event." Thus, amended independent claim 28 is patentable over the cited art.

With respect independent claim 1, this claim is a method claim that recited elements similar to the system claim of claim 28, has been amended in a manner

analogous to amended claim 28. Accordingly, amended claim 1 is also patentable over the cited art for similar reasons to claim 28.

With respect to independent claim 13, this claim does not recite elements that are analogous to similar elements in claim 28. Claim 13 recites:

13. A method for extending a System Management Mode (SMM) of a microprocessor in a computer system, comprising:

publishing an interface during a pre-boot process for the computer system to enable a driver that is stored outside of components in which the original set of firmware is stored to provide a set of machine code comprising an event handler that is loaded into SMM memory (SMRAM) that is accessible to the microprocessor while in SMM;

switching the microprocessor to SMM in response to an SMM triggering event; and

executing the event handler to service the SMM triggering event.

Clearly, neither *Kimura* or *Datta* teach or suggest the elements in the first sub-paragraph of claim 13. Neither reference publishes an interface during the pre-boot process of a computer system, nor enable a driver to be loaded into SMRAM as in event handler. Accordingly, claim 13 is patentable over the cited art.

With respect to independent claim 23, this claim has been amended to now recite:

23. A machine-readable medium having a plurality of machine instructions stored thereon that when executed by a processor in a computer system performs the operations of:

providing a mechanism to enable a loading of a plurality of event handlers that are not stored in an original set of firmware as supplied by an original equipment manufacture (OEM) of the computer system into a hidden memory space that is accessible to a hidden execution and storage mode of the processor but is not accessible to other operating modes of the processor, *the mechanism enabling event*

handlers from at least two different third party sources to be loaded into the hidden memory space; and

selectively executing an appropriate event handler from among the plurality of event handlers in response to an event that causes the processor to be switched to the hidden execution and storage mode to service the event.

Claim 23 is directed to *machine instructions* (*e.g.*, firmware or software code or modules) that are executed to perform the recited operations. Notably, the claim recites that the mechanisms enables event handlers from at least two different third party sources to be loaded into the hidden memory space. Thus, the mechanism provides an extensible framework that enables event handlers written by different parties to be loaded and used for SMI event handling purposes.

This is clearly not possible with *Kimura*. As discussed above, *Kimura*'s scheme redirects the SMI processing to its SMM handler, which comprises a monolithic block of code with no extensibility for supporting other SMM handlers or event handling modules. In fact, *Kimura*'s scheme is dependent, in part, on hardware modifications to the computer in which the scheme is implemented. Specifically, *Kimura* employs a hardware mechanism for identifying the event that needs to be serviced, using registers contained in an add-on hardware component. As shown in FIG. 8A and FIG. 8B, *Kimura* employs groups of event registers to identify which event to service. The event registers are facilitated by hardware shown in FIGs. 9A-B, 10A-B, 11A-C, and 12A-B. Clearly, determination of what event to service is made by must be ascertained from this hardware, and not determined by firmware or software on its own. The *Kimura* software thus needs to be aware of the hardware register configuration, and is written in view of an expected configuration. This would prevent third-party extensibility, especially in regard to supporting third-party peripheral add-on boards that employ their own sets of registers for identifying SMI events. Accordingly, the rejection of claim 23 is improper and should be withdrawn.

As discussed above, many of the claims contained in claim 1-27 do not pertain to analogous claims in claim 28-30. These include claim 4, 6-12, 14-22, and 26.

With further respect to dependent claim 29, (as well as dependent claims 4 and 24), neither *Datta* or *Kimura* teach of suggest, alone or in combination,

"providing an abstracted interface that enables a set of machine code corresponding to an event handler that is stored outside of any component(s) in which the original set of firmware is stored to be loaded into the hidden memory space."

The recited text (abstract and col. 69, lines 30-39, presented above) does not support any abstracted interface whatsoever. The abstracted interface, which is somewhat akin to an application program interface (API) used for software applications, enables event handlers from different sources to be loaded in the hidden memory space using a predefined interface. The interface "abstracts" the underlying mechanisms that facilitate SMI event handling. This is clearly not taught by *Kimura*. Accordingly, the rejection of claim 28 (as well as claims 4 and 24) is improper and should be withdrawn.

With respect to claim 30 (as well as claims 5, 18 and 25), *Datta* clearly does not teach "scanning for any firmware volumes that are materialized during a pre-boot process for the computer system to identify an existence of any firmware file containing an event handler that is compatible with the hidden execution and storage mode of the processor." The concept of firmware volumes, which was developed by Intel corporation, the assignee of the *Datta* patent, was not developed until approximately 2000, which is five years subsequent to when the *Datta* patent was filed. In fact, both the Applicant (Vincent Zimmer) and Sham *Datta* worked for the engineering group that developed the concept of firmware volumes at Intel at the time the present application was filed in.

With further respect to independent claim 19, this claim recites claim elements that are not analogous to claims 28-30, and thus were not addressed by the present

Office Action. Applicants respectfully assert that independent claim 19 is clearly patentable over the cited art.

Overall, none of the references singly or in any motivated combination disclose, teach, or suggest what is recited in the independent claims. Thus, given the above amendments and accompanying remarks, each of independent claims 1, 13, 19, 23, and 28 are now in condition for allowance. The dependent claims that depend directly or indirectly on these independent claims are likewise allowable based on at least the same reasons and based on the recitations contained in each dependent claim.

If the undersigned attorney has overlooked a teaching in any of the cited references that is relevant to the allowability of the claims, the Examiner is requested to specifically point out where such teaching may be found. Further, if there are any informalities or questions that can be addressed via telephone, the Examiner is encouraged to contact the undersigned attorney at (206) 292-8600.

Charge Deposit Account

Please charge our Deposit Account No. 02-2666 for any additional fee(s) that may be due in this matter, and please credit the same deposit account for any overpayment.

Respectfully submitted,
BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN

Date: Aug 5, 2004

R. Alan Burnett
Reg. No. 46,149

12400 Wilshire Boulevard
Seventh Floor
Los Angeles, CA 90025-1026

Enclosures: Intel System Management document

IA-32 Intel® Architecture Software Developer's Manual

Volume 3: System Programming Guide

NOTE: The *IA-32 Intel Architecture Developer's Manual* consists of three books: *Basic Architecture*, Order Number 245470-012; *Instruction Set Reference Manual*, Order Number 245471-012; and the *System Programming Guide*, Order Number 245472-012. Please refer to all three volumes when evaluating your design needs.

2003

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature, may be obtained from:

Intel Corporation
P.O. Box 5937
Denver, CO 80217-9808

or call 1-800-548-4725
or visit Intel's website at http://www.intel.com

intel®

CHAPTER 13
SYSTEM MANAGEMENT

This chapter describes the two aspects of IA-32 architecture used to manage system resources: system management mode (SMM) and the thermal monitoring facilities.

SMM provides an alternate operating environment that can be used to monitor and manage various system resources for more efficient energy usage, to control system hardware, and/or to run proprietary code. It was introduced into the IA-32 architecture in the Intel386 SL processor (a mobile specialized version of the Intel386 processor). It is also available in the Pentium 4, Intel Xeon, P6 family, and Pentium and Intel486 processors (beginning with the enhanced versions of the Intel486 SL and Intel486 processors). For a detailed description of the hardware that supports SMM, see the developer's manuals for each of the IA-32 processors.

The thermal monitoring facilities enable monitoring and controlling the core temperature of an IA-32 processor. These facilities were introduced in the P6 family processors and extended in the Pentium 4 , Intel Xeon and Pentium M processors.

13.1. SYSTEM MANAGEMENT MODE OVERVIEW

SMM is a special-purpose operating mode provided for handling system-wide functions like power management, system hardware control, or proprietary OEM-designed code. It is intended for use only by system firmware, not by applications software or general-purpose systems software. The main benefit of SMM is that it offers a distinct and easily isolated processor environment that operates transparently to the operating system or executive and software applications.

When SMM is invoked through a system management interrupt (SMI), the processor saves the current state of the processor (the processor's context), then switches to a separate operating environment contained in system management RAM (SMRAM). While in SMM, the processor executes SMI handler code to perform operations such as powering down unused disk drives or monitors, executing proprietary code, or placing the whole system in a suspended state. When the SMI handler has completed its operations, it executes a resume (RSM) instruction. This instruction causes the processor to reload the saved context of the processor, switch back to protected or real mode, and resume executing the interrupted application or operating-system program or task.

The following SMM mechanisms make it transparent to applications programs and operating systems:

- The only way to enter SMM is by means of an SMI.
- The processor executes SMM code in a separate address space (SMRAM) that can be made inaccessible from the other operating modes.
- Upon entering SMM, the processor saves the context of the interrupted program or task.

- All interrupts normally handled by the operating system are disabled upon entry into SMM.
- The RSM instruction can be executed only in SMM.

SMM is similar to real-address mode in that there are no privilege levels or address mapping. An SMM program can address up to 4 GBytes of memory and can execute all I/O and applicable system instructions. See Section 13.5., "SMI Handler Execution Environment", for more information about the SMM execution environment.

NOTE

The physical address extension (PAE) mechanism available in the P6 family processors is not supported when a processor is in SMM.

13.2. SYSTEM MANAGEMENT INTERRUPT (SMI)

The only way to enter SMM is by signaling an SMI through the SMI# pin on the processor or through an SMI message received through the APIC bus. The SMI is a nonmaskable external interrupt that operates independently from the processor's interrupt- and exception-handling mechanism and the local APIC. The SMI takes precedence over an NMI and a maskable interrupt. SMM is non-reentrant; that is, the SMI is disabled while the processor is in SMM.

NOTE

In the Pentium 4, Intel Xeon, and P6 family processors, when a processor that is designated as an application processor during an MP initialization sequence is waiting for a startup IPI (SIPI), it is in a mode where SMIs are masked. However if a SMI is received while an application processor is in the wait for SIPI mode, the SMI will be pended. The processor then responds on receipt of a SIPI by immediately servicing the pended SMI and going into SMM before handling the SIPI.

13.3. SWITCHING BETWEEN SMM AND THE OTHER PROCESSOR OPERATING MODES

Figure 2-2 shows how the processor moves between SMM and the other processor operating modes (protected, real-address, and virtual-8086). Signaling an SMI while the processor is in real-address, protected, or virtual-8086 modes always causes the processor to switch to SMM. Upon execution of the RSM instruction, the processor always returns to the mode it was in when the SMI occurred.

13.3.1. Entering SMM

The processor always handles an SMI on an architecturally defined "interruptible" point in program execution (which is commonly at an IA-32 architecture instruction boundary). When the processor receives an SMI, it waits for all instructions to retire and for all stores to complete. The processor then saves its current context in SMRAM (see Section 13.4., "SMRAM"), enters SMM, and begins to execute the SMI handler.

Upon entering SMM, the processor signals external hardware that SMM handling has begun. The signaling mechanism used is implementation dependent. For the P6 family processors, an SMI acknowledge transaction is generated on the system bus and the multiplexed status signal EXF4 is asserted each time a bus transaction is generated while the processor is in SMM. For the Pentium and Intel486 processors, the SMIACT# pin is asserted.

An SMI has a greater priority than debug exceptions and external interrupts. Thus, if an NMI, maskable hardware interrupt, or a debug exception occurs at an instruction boundary along with an SMI, only the SMI is handled. Subsequent SMI requests are not acknowledged while the processor is in SMM. The first SMI interrupt request that occurs while the processor is in SMM (that is, after SMM has been acknowledged to external hardware) is latched and serviced when the processor exits SMM with the RSM instruction. The processor will latch only one SMI while in SMM.

See Section 13.5., "SMI Handler Execution Environment", for a detailed description of the execution environment when in SMM.

13.3.2. Exiting From SMM

The only way to exit SMM is to execute the RSM instruction. The RSM instruction is only available to the SMI handler; if the processor is not in SMM, attempts to execute the RSM instruction result in an invalid-opcode exception (#UD) being generated.

The RSM instruction restores the processor's context by loading the state save image from SMRAM back into the processor's registers. The processor then returns an SMIACK transaction on the system bus and returns program control back to the interrupted program.

Upon successful completion of the RSM instruction, the processor signals external hardware that SMM has been exited. For the P6 family processors, an SMI acknowledge transaction is generated on the system bus and the multiplexed status signal EXF4 is no longer generated on bus cycles. For the Pentium and Intel486 processors, the SMIACT# pin is deserted.

If the processor detects invalid state information saved in the SMRAM, it enters the shutdown state and generates a special bus cycle to indicate it has entered shutdown state. Shutdown happens only in the following situations:

- A reserved bit in control register CR4 is set to 1 on a write to CR4. This error should not happen unless SMI handler code modifies reserved areas of the SMRAM saved state map (see Section 13.4.1., "SMRAM State Save Map"). Note that CR4 is saved in the state map in a reserved location and cannot be read or modified in its saved state.

- An illegal combination of bits is written to control register CR0, in particular PG set to 1 and PE set to 0, or NW set to 1 and CD set to 0.
- (For the Pentium and Intel486 processors only.) If the address stored in the SMBASE register when an RSM instruction is executed is not aligned on a 32-KByte boundary. This restriction does not apply to the P6 family processors.

In the shutdown state, Intel processors stop executing instructions until a RESET#, INIT# or NMI# is asserted. Processors do recognize the FLUSH# signal in the shutdown state. While Pentium family processors recognize the SMI# signal in shutdown state, P6 family and Intel486 processors do not. Intel does not support using SMI# to recover from shutdown states for any processor family; the response of processors in this circumstance is not well defined.

If the processor is in the HALT state when the SMI is received, the processor handles the return from SMM slightly differently (see Section 13.10., "Auto HALT Restart"). Also, the SMBASE address can be changed on a return from SMM (see Section 13.11., "SMBASE Relocation").

13.4. SMRAM

While in SMM, the processor executes code and stores data in the SMRAM space. The SMRAM space is mapped to the physical address space of the processor and can be up to 4 GBytes in size. The processor uses this space to save the context of the processor and to store the SMI handler code, data and stack. It can also be used to store system management information (such as the system configuration and specific information about powered-down devices) and OEM-specific information.

The default SMRAM size is 64 KBytes beginning at a base physical address in physical memory called the SMBASE (see Figure 13-1). The SMBASE default value following a hardware reset is 30000H. The processor looks for the first instruction of the SMI handler at the address [SMBASE + 8000H]. It stores the processor's state in the area from [SMBASE + FE00H] to [SMBASE + FFFFH]. See Section 13.4.1., "SMRAM State Save Map", for a description of the mapping of the state save area.

The system logic is minimally required to decode the physical address range for the SMRAM from [SMBASE + 8000H] to [SMBASE + FFFFH]. A larger area can be decoded if needed. The size of this SMRAM can be between 32 KBytes and 4 GBytes.

The location of the SMRAM can be changed by changing the SMBASE value (see Section 13.11., "SMBASE Relocation"). It should be noted that all processors in a multiple-processor system are initialized with the same SMBASE value (30000H). Initialization software must sequentially place each processor in SMM and change its SMBASE so that it does not overlap those of other processors.

The actual physical location of the SMRAM can be in system memory or in a separate RAM memory. The processor generates an SMI acknowledge transaction (P6 family processors) or asserts the SMIACT# pin (Pentium and Intel486 processors) when the processor receives an SMI (see Section 13.3.1., "Entering SMM").

System logic can use the SMI acknowledge transaction or the assertion of the SMIACT# pin to decode accesses to the SMRAM and redirect them (if desired) to specific SMRAM memory. If

a separate RAM memory is used for SMRAM, system logic should provide a programmable method of mapping the SMRAM into system memory space when the processor is not in SMM. This mechanism will enable start-up procedures to initialize the SMRAM space (that is, load the SMI handler) before executing the SMI handler during SMM.

13.4.1. SMRAM State Save Map

When the processor initially enters SMM, it writes its state to the state save area of the SMRAM. The state save area begins at [SMBASE + 8000H + 7FFFH] and extends down to [SMBASE + 8000H + 7E00H]. Table 13-1 shows the state save map. The offset in column 1 is relative to the SMBASE value plus 8000H. Reserved spaces should not be used by software.

Some of the registers in the SMRAM state save area (marked YES in column 3) may be read and changed by the SMI handler, with the changed values restored to the processor registers by the RSM instruction. Some register images are read-only, and must not be modified (modifying these registers will result in unpredictable behavior). An SMI handler should not rely on any values stored in an area that is marked as reserved.

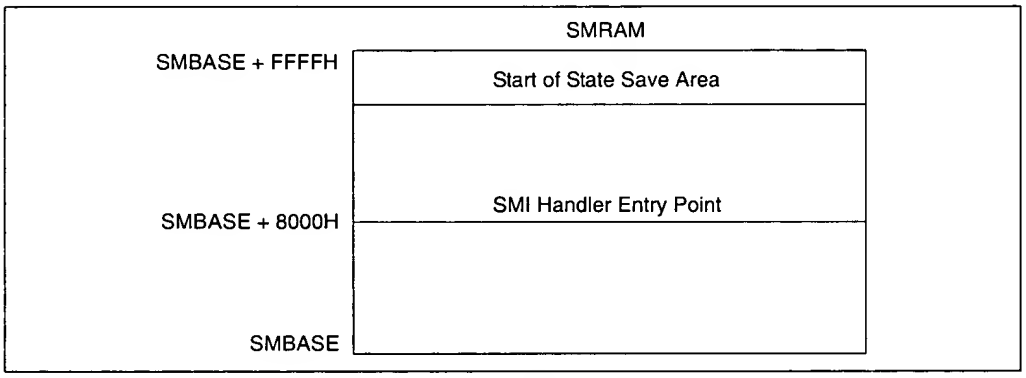

Figure 13-1. SMRAM Usage

Table 13-1. SMRAM State Save Map

Offset (Added to SMBASE + 8000H)	Register	Writable?
7FFCH	CR0	No
7FF8H	CR3	No
7FF4H	EFLAGS	Yes
7FF0H	EIP	Yes
7FECH	EDI	Yes
7FE8H	ESI	Yes
7FE4H	EBP	Yes

Table 13-1. SMRAM State Save Map (Contd.)

Offset (Added to SMBASE + 8000H)	Register	Writable?
7FE0H	ESP	Yes
7FDCH	EBX	Yes
7FD8H	EDX	Yes
7FD4H	ECX	Yes
7FD0H	EAX	Yes
7FCCH	DR6	No
7FC8H	DR7	No
7FC4H	TR*	No
7FC0H	Reserved	No
7FBCH	GS*	No
7FB8H	FS*	No
7FB4H	DS*	No
7FB0H	SS*	No
7FACH	CS*	No
7FA8H	ES*	No
7FA7H - 7F98H	Reserved	No
7F94H	Reserved	No
7F93H - 7F8CH	Reserved	No
7F88H	Reserved	No
7F87H - 7F04H	Reserved	No
7F02H	Auto HALT Restart Field (Word)	Yes
7F00H	I/O Instruction Restart Field (Word)	Yes
7EFCH	SMM Revision Identifier Field (Doubleword)	No
7EF8H	SMBASE Field (Doubleword)	Yes
7EF7H - 7E00H	Reserved	No

NOTE:

* Upper two bytes are reserved.

The following registers are saved (but not readable) and restored upon exiting SMM:

- Control register CR4. (This register is cleared to all 0s while in SMM).
- The hidden segment descriptor information stored in segment registers CS, DS, ES, FS, GS, and SS.

If an SMI request is issued for the purpose of powering down the processor, the values of all reserved locations in the SMM state save must be saved to nonvolatile memory.

The following state is not automatically saved and restored following an SMI and the RSM instruction, respectively:

- Debug registers DR0 through DR3.
- The x87 FPU registers.
- The MTRRs.
- Control register CR2.
- The model-specific registers (for the P6 family and Pentium processors) or test registers TR3 through TR7 (for the Pentium and Intel486 processors).
- The state of the trap controller.
- The machine-check architecture registers.
- The APIC internal interrupt state (ISR, IRR, etc.).
- The microcode update state.

If an SMI is used to power down the processor, a power-on reset will be required before returning to SMM, which will reset much of this state back to its default values. So an SMI handler that is going to trigger power down should first read these registers listed above directly, and save them (along with the rest of RAM) to nonvolatile storage. After the power-on reset, the continuation of the SMI handler should restore these values, along with the rest of the system's state. Anytime the SMI handler changes these registers in the processor, it must also save and restore them.

NOTE

A small subset of the MSRs (such as, the time-stamp counter and performance-monitoring counters) are not arbitrarily writable and therefore cannot be saved and restored. SMM-based power-down and restoration should only be performed with operating systems that do not use or rely on the values of these registers. Operating system developers should be aware of this fact and insure that their operating-system assisted power-down and restoration software is immune to unexpected changes in these register values.

13.4.2. SMRAM Caching

An IA-32 processor does not automatically write back and invalidate its caches before entering SMM or before exiting SMM. Because of this behavior, care must be taken in the placement of the SMRAM in system memory and in the caching of the SMRAM to prevent cache incoherence when switching back and forth between SMM and protected mode operation. Either of the following three methods of locating the SMRAM in system memory will guarantee cache coherency:

- Place the SRAM in a dedicated section of system memory that the operating system and applications are prevented from accessing. Here, the SRAM can be designated as cacheable (WB, WT, or WC) for optimum processor performance, without risking cache incoherence when entering or exiting SMM.
- Place the SRAM in a section of memory that overlaps an area used by the operating system (such as the video memory), but designate the SMRAM as uncacheable (UC). This method prevents cache access when in SMM to maintain cache coherency, but the use of uncacheable memory reduces the performance of SMM code.
- Place the SRAM in a section of system memory that overlaps an area used by the operating system and/or application code, but explicitly flush (write back and invalidate) the caches upon entering and exiting SMM mode. This method maintains cache coherency, but the incurs the overhead of two complete cache flushes.

For Pentium 4, Intel Xeon, and P6 family processors, a combination of the first two methods of locating the SMRAM is recommended. Here the SMRAM is split between an overlapping and a dedicated region of memory. Upon entering SMM, the SMRAM space that is accessed overlaps video memory (typically located in low memory). This SMRAM section is designated as UC memory. The initial SMM code then jumps to a second SMRAM section that is located in a dedicated region of system memory (typically in high memory). This SMRAM section can be cached for optimum processor performance.

For systems that explicitly flush the caches upon entering SMM (the third method described above), the cache flush can be accomplished by asserting the FLUSH# pin at the same time as the request to enter SMM (generally initiated by asserting the SMI# pin). The priorities of the FLUSH# and SMI# pins are such that the FLUSH# is serviced first. To guarantee this behavior, the processor requires that the following constraints on the interaction of FLUSH# and SMI# be met. In a system where the FLUSH# and SMI# pins are synchronous and the set up and hold times are met, then the FLUSH# and SMI# pins may be asserted in the same clock. In asynchronous systems, the FLUSH# pin must be asserted at least one clock before the SMI# pin to guarantee that the FLUSH# pin is serviced first.

Upon leaving SMM (for systems that explicitly flush the caches), the WBINVD instruction should be executed prior to leaving SMM to flush the caches.

NOTE

In systems based on the Pentium processor that use the FLUSH# pin to write back and invalidate cache contents before entering SMM, the processor will prefetch at least one cache line in between when the Flush Acknowledge cycle is run, and the subsequent recognition of SMI# and the assertion of SMIACT#. It is the obligation of the system to ensure that these lines are not cached by returning KEN# inactive to the Pentium processor.

13.5. SMI HANDLER EXECUTION ENVIRONMENT

After saving the current context of the processor, the processor initializes its core registers to the values shown in Table 13-2. Upon entering SMM, the PE and PG flags in control register CR0 are cleared, which places the processor is in an environment similar to real-address mode. The differences between the SMM execution environment and the real-address mode execution environment are as follows:

- The addressable SMRAM address space ranges from 0 to FFFFFFFFH (4 GBytes). (The physical address extension (enabled with the PAE flag in control register CR4) is not supported in SMM.)
- The normal 64-KByte segment limit for real-address mode is increased to 4 GBytes.
- The default operand and address sizes are set to 16 bits, which restricts the addressable SMRAM address space to the 1-MByte real-address mode limit for native real-address-mode code. However, operand-size and address-size override prefixes can be used to access the address space beyond the 1-MByte.

Table 13-2. Processor Register Initialization in SMM

Register	Contents
General-purpose registers	Undefined
EFLAGS	00000002H
EIP	00008000H
CS selector	SMM Base shifted right 4 bits (default 3000H)
CS base	SMM Base (default 30000H)
DS, ES, FS, GS, SS Selectors	0000H
DS, ES, FS, GS, SS Bases	000000000H
DS, ES, FS, GS, SS Limits	0FFFFFFFFH
CR0	PE, EM, TS and PG flags set to 0; others unmodified
CR4	Cleared to zero
DR6	Undefined
DR7	00000400H

- Near jumps and calls can be made to anywhere in the 4-GByte address space if a 32-bit operand-size override prefix is used. Due to the real-address-mode style of base-address formation, a far call or jump cannot transfer control to a segment with a base address of more than 20 bits (1 MByte). However, since the segment limit in SMM is 4 GBytes, offsets into a segment that go beyond the 1-MByte limit are allowed when using 32-bit operand-size override prefixes. Any program control transfer that does not have a 32-bit operand-size override prefix truncates the EIP value to the 16 low-order bits.
- Data and the stack can be located anywhere in the 4-GByte address space, but can be accessed only with a 32-bit address-size override if they are located above 1 MByte. As with the code segment, the base address for a data or stack segment cannot be more than 20 bits.

The value in segment register CS is automatically set to the default of 30000H for the SMBASE shifted 4 bits to the right; that is, 3000H. The EIP register is set to 8000H. When the EIP value is added to shifted CS value (the SMBASE), the resulting linear address points to the first instruction of the SMI handler.

The other segment registers (DS, SS, ES, FS, and GS) are cleared to 0 and their segment limits are set to 4 GBytes. In this state, the SMRAM address space may be treated as a single flat 4-Gbyte linear address space. If a segment register is loaded with a 16-bit value, that value is then shifted left by 4 bits and loaded into the segment base (hidden part of the segment register). The limits and attributes are not modified.

Maskable hardware interrupts, exceptions, NMI interrupts, SMI interrupts, A20M interrupts, single-step traps, breakpoint traps, and INIT operations are inhibited when the processor enters SMM. Maskable hardware interrupts, exceptions, single-step traps, and breakpoint traps can be enabled in SMM if the SMM execution environment provides and initializes an interrupt table and the necessary interrupt and exception handlers (see Section 13.6., "Exceptions and Interrupts Within SMM").

13.6. EXCEPTIONS AND INTERRUPTS WITHIN SMM

When the processor enters SMM, all hardware interrupts are disabled in the following manner:

- The IF flag in the EFLAGS register is cleared, which inhibits maskable hardware interrupts from being generated.
- The TF flag in the EFLAGS register is cleared, which disables single-step traps.
- Debug register DR7 is cleared, which disables breakpoint traps. (This action prevents a debugger from accidentally breaking into an SMM handler if a debug breakpoint is set in normal address space that overlays code or data in SMRAM.)
- NMI, SMI, and A20M interrupts are blocked by internal SMM logic. (See Section 13.7., "NMI Handling While in SMM", for further information about how NMIs are handled in SMM.)

Software-invoked interrupts and exceptions can still occur, and maskable hardware interrupts can be enabled by setting the IF flag. Intel recommends that SMM code be written in so that it does not invoke software interrupts (with the INT *n*, INTO, INT 3, or BOUND instructions) or generate exceptions.

If the SMM handler requires interrupt and exception handling, an SMM interrupt table and the necessary exception and interrupt handlers must be created and initialized from within SMM. Until the interrupt table is correctly initialized (using the LIDT instruction), exceptions and software interrupts will result in unpredictable processor behavior.

The following restrictions apply when designing SMM interrupt and exception-handling facilities:

- The interrupt table should be located at linear address 0 and must contain real-address mode style interrupt vectors (4 bytes containing CS and IP).
- Due to the real-address mode style of base address formation, an interrupt or exception cannot transfer control to a segment with a base address of more that 20 bits.
- An interrupt or exception cannot transfer control to a segment offset of more than 16 bits (64 KBytes).
- When an exception or interrupt occurs, only the 16 least-significant bits of the return address (EIP) are pushed onto the stack. If the offset of the interrupted procedure is greater than 64 KBytes, it is not possible for the interrupt/exception handler to return control to that procedure. (One solution to this problem is for a handler to adjust the return address on the stack.)
- The SMBASE relocation feature affects the way the processor will return from an interrupt or exception generated while the SMI handler is executing. For example, if the SMBASE is relocated to above 1 MByte, but the exception handlers are below 1 MByte, a normal return to the SMI handler is not possible. One solution is to provide the exception handler with a mechanism for calculating a return address above 1 MByte from the 16-bit return address on the stack, then use a 32-bit far call to return to the interrupted procedure.
- If an SMI handler needs access to the debug trap facilities, it must insure that an SMM accessible debug handler is available and save the current contents of debug registers DR0 through DR3 (for later restoration). Debug registers DR0 through DR3 and DR7 must then be initialized with the appropriate values.
- If an SMI handler needs access to the single-step mechanism, it must insure that an SMM accessible single-step handler is available, and then set the TF flag in the EFLAGS register.
- If the SMI design requires the processor to respond to maskable hardware interrupts or software-generated interrupts while in SMM, it must ensure that SMM accessible interrupt handlers are available and then set the IF flag in the EFLAGS register (using the STI instruction). Software interrupts are not blocked upon entry to SMM, so they do not need to be enabled.

13.7. NMI HANDLING WHILE IN SMM

NMI interrupts are blocked upon entry to the SMI handler. If an NMI request occurs during the SMI handler, it is latched and serviced after the processor exits SMM. Only one NMI request will be latched during the SMI handler. If an NMI request is pending when the processor executes the RSM instruction, the NMI is serviced before the next instruction of the interrupted code sequenc. This assumes that NMIs were not blocked before the SMI occurred. If NMIs were blocked before the SMI occurred, they are blocked after execution of RSM.

Although NMI requests are blocked when the processor enters SMM, they may be enabled through software by executing an IRET/IRETD instruction. If the SMM handler requires the use of NMI interrupts, it should invoke a dummy interrupt service routine for the purpose of executing an IRET/IRETD instruction. Once an IRET/IRETD instruction is executed, NMI interrupt requests are serviced in the same "real mode" manner in which they are handled outside of SMM.

A special case can occur if an SMI handler nests inside an NMI handler and then another NMI occurs. During NMI interrupt handling, NMI interrupts are disabled, so normally NMI interrupts are serviced and completed with an IRET instruction one at a time. When the processor enters SMM while executing an NMI handler, the processor saves the SMRAM state save map but does not save the attribute to keep NMI interrupts disabled. Potentially, an NMI could be latched (while in SMM or upon exit) and serviced upon exit of SMM even though the previous NMI handler has still not completed. One or more NMIs could thus be nested inside the first NMI handler. The NMI interrupt handler should take this possibility into consideration.

Also, for the Pentium processor, exceptions that invoke a trap or fault handler will enable NMI interrupts from inside of SMM. This behavior is implementation specific for the Pentium processor and is not part the IA-32 architecture.

13.8. SAVING THE X87 FPU STATE WHILE IN SMM

In some instances (for example prior to powering down system memory when entering a 0-volt suspend state), it is necessary to save the state of the x87 FPU while in SMM. Care should be taken when performing this operation to insure that relevant x87 FPU state information is not lost. The safest way to perform this task is to place the processor in 32-bit protected mode before saving the x87 FPU state. The reason for this is as follows.

The FSAVE instruction saves the x87 FPU context in any of four different formats, depending on which mode the processor is in when FSAVE is executed (see Figures 8-9 through 8-12 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*). When in SMM, by default, the 16-bit real-address mode format is used (shown in Figure 8-12). If an SMI interrupt occurs while the processor is in a mode other than 16-bit real-address mode, FSAVE and FRSTOR will be unable to save and restore all the relevant x87 FPU information, and this situation may result in a malfunction when the interrupted program is resumed. To avoid this problem, the processor should be in 32-bit protected mode when executing the FSAVE and FRSTOR instructions.

The following guidelines should be used when going into protected mode from an SMI handler to save and restore the x87 FPU state:

- Use the CPUID instruction to insure that the processor contains an x87 FPU.
- Create a 32-bit code segment in SMRAM space that contains procedures or routines to save and restore the x87 FPU using the FSAVE and FRSTOR instructions, respectively. A GDT with an appropriate code-segment descriptor (D bit is set to 1) for the 32-bit code segment must also be placed in SMRAM.
- Write a procedure or routine that can be called by the SMI handler to save and restore the x87 FPU state. This procedure should do the following:
  - — Place the processor in 32-bit protected mode as describe in Section 9.9.1., "Switching to Protected Mode".
  - — Execute a far JMP to the 32-bit code segment that contains the x87 FPU save and restore procedures.
  - — Place the processor back in 16-bit real-address mode before returning to the SMI handler (see Section 9.9.2., "Switching Back to Real-Address Mode").

The SMI handler may continue to execute in protected mode after the x87 FPU state has been saved and return safely to the interrupted program from protected mode. However, it is recommended that the handler execute primarily in 16- or 32-bit real-address mode.

13.9. SMM REVISION IDENTIFIER

The SMM revision identifier field is used to indicate the version of SMM and the SMM extensions that are supported by the processor (see Figure 13-2). The SMM revision identifier is written during SMM entry and can be examined in SMRAM space at offset 7EFCH. The lower word of the SMM revision identifier refers to the version of the base SMM architecture.

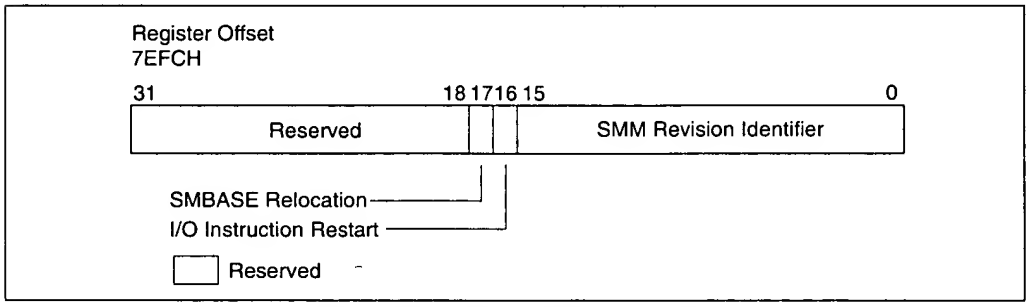

Figure 13-2. SMM Revision Identifier

The upper word of the SMM revision identifier refers to the extensions available. If the I/O instruction restart flag (bit 16) is set, the processor supports the I/O instruction restart (see Section 13.12., "I/O Instruction Restart"); if the SMBASE relocation flag (bit 17) is set, SMRAM base address relocation is supported (see Section 13.11., "SMBASE Relocation").

13.10. AUTO HALT RESTART

If the processor is in a HALT state (due to the prior execution of a HLT instruction) when it receives an SMI, the processor records the fact in the auto HALT restart flag in the saved processor state (see Figure 13-3). (This flag is located at offset 7F02H and bit 0 in the state save area of the SMRAM.)

If the processor sets the auto HALT restart flag upon entering SMM (indicating that the SMI occurred when the processor was in the HALT state), the SMI handler has two options:

- It can leave the auto HALT restart flag set, which instructs the RSM instruction to return program control to the HLT instruction. This option in effect causes the processor to re-enter the HALT state after handling the SMI. (This is the default operation.)
- It can clear the auto HALT restart flag, with instructs the RSM instruction to return program control to the instruction following the HLT instruction.

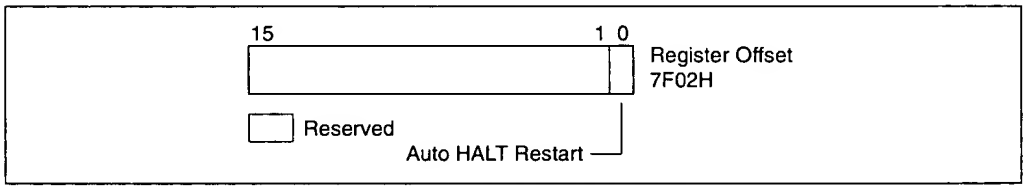

Figure 13-3. Auto HALT Restart Field

These options are summarized in Table 13-3. Note that if the processor was not in a HALT state when the SMI was received (the auto HALT restart flag is cleared), setting the flag to 1 will cause unpredictable behavior when the RSM instruction is executed.

Table 13-3. Auto HALT Restart Flag Values

Value of Flag After Entry to SMM	Value of Flag When Exiting SMM	Action of Processor When Exiting SMM
0	0	Returns to next instruction in interrupted program or task
0	1	Unpredictable
1	0	Returns to next instruction after HLT instruction
1	1	Returns to HALT state

If the HLT instruction is restarted, the processor will generate a memory access to fetch the HLT instruction (if it is not in the internal cache), and execute a HLT bus transaction. This behavior results in multiple HLT bus transactions for the same HLT instruction.

13.10.1. Executing the HLT Instruction in SMM

The HLT instruction should not be executed during SMM, unless interrupts have been enabled by setting the IF flag in the EFLAGS register. If the processor is halted in SMM, the only event that can remove the processor from this state is a maskable hardware interrupt or a hardware reset.

13.11. SMBASE RELOCATION

The default base address for the SMRAM is 30000H. This value is contained in an internal processor register called the SMBASE register. The operating system or executive can relocate the SMRAM by setting the SMBASE field in the saved state map (at offset 7EF8H) to a new value (see Figure 13-4). The RSM instruction reloads the internal SMBASE register with the value in the SMBASE field each time it exits SMM. All subsequent SMI requests will use the new SMBASE value to find the starting address for the SMI handler (at SMBASE + 8000H) and the SMRAM state save area (from SMBASE + FE00H to SMBASE + FFFFH). (The processor resets the value in its internal SMBASE register to 30000H on a RESET, but does not change it on an INIT.)

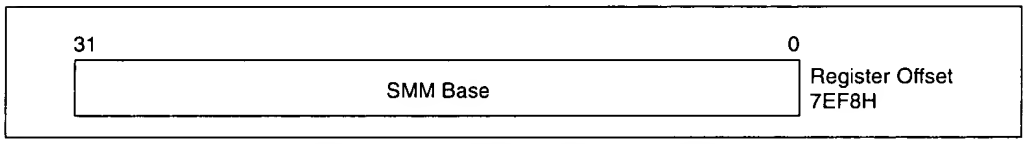

Figure 13-4. SMBASE Relocation Field

In multiple-processor systems, initialization software must adjust the SMBASE value for each processor so that the SMRAM state save areas for each processor do not overlap. (For Pentium and Intel486 processors, the SMBASE values must be aligned on a 32-KByte boundary or the processor will enter shutdown state during the execution of a RSM instruction.)

If the SMBASE relocation flag in the SMM revision identifier field is set, it indicates the ability to relocate the SMBASE (see Section 13.9., "SMM Revision Identifier").

13.11.1. Relocating SMRAM to an Address Above 1 MByte

In SMM, the segment base registers can only be updated by changing the value in the segment registers. The segment registers contain only 16 bits, which allows only 20 bits to be used for a segment base address (the segment register is shifted left 4 bits to determine the segment base address). If SMRAM is relocated to an address above 1 MByte, software operating in real-address mode can no longer initialize the segment registers to point to the SMRAM base address (SMBASE).

The SMRAM can still be accessed by using 32-bit address-size override prefixes to generate an offset to the correct address. For example, if the SMBASE has been relocated to FFFFFFH (immediately below the 16-MByte boundary) and the DS, ES, FS, and GS registers are still initialized to 0H, data in SMRAM can be accessed by using 32-bit displacement registers, as in the following example:

```
mov    esi,00FFxxxxH; 64K segment immediately below 16M
mov    ax,ds:[esi]
```

A stack located above the 1-MByte boundary can be accessed in the same manner.

13.12. I/O INSTRUCTION RESTART

If the I/O instruction restart flag in the SMM revision identifier field is set (see Section 13.9., "SMM Revision Identifier"), the I/O instruction restart mechanism is present on the processor. This mechanism allows an interrupted I/O instruction to be re-executed upon returning from SMM mode. For example, if an I/O instruction is used to access a powered-down I/O device, a chip set supporting this device can intercept the access and respond by asserting SMI#. This action invokes the SMI handler to power-up the device. Upon returning from the SMI handler, the I/O instruction restart mechanism can be used to re-execute the I/O instruction that caused the SMI.

The I/O instruction restart field (at offset 7F00H in the SMM state-save area, see Figure 13-5) controls I/O instruction restart. When an RSM instruction is executed, if this field contains the value FFH, then the EIP register is modified to point to the I/O instruction that received the SMI request. The processor will then automatically re-execute the I/O instruction that the SMI trapped. (The processor saves the necessary machine state to insure that re-execution of the instruction is handled coherently.)

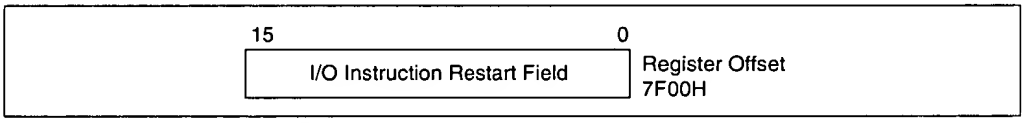

Figure 13-5. I/O Instruction Restart Field

If the I/O instruction restart field contains the value 00H when the RSM instruction is executed, then the processor begins program execution with the instruction following the I/O instruction. (When a repeat prefix is being used, the next instruction may be the next I/O instruction in the repeat loop.) Not re-executing the interrupted I/O instruction is the default behavior; the processor automatically initializes the I/O instruction restart field to 00H upon entering SMM. Table 13-4 summarizes the states of the I/O instruction restart field.

Table 13-4. I/O Instruction Restart Field Values

Value of Flag After Entry to SMM	Value of Flag When Exiting SMM	Action of Processor When Exiting SMM
00H	00H	Does not re-execute trapped I/O instruction.
00H	FFH	Re-executes trapped I/O instruction.

Note that the I/O instruction restart mechanism does not indicate the cause of the SMI. It is the responsibility of the SMI handler to examine the state of the processor to determine the cause of the SMI and to determine if an I/O instruction was interrupted and should be restarted upon exiting SMM. If an SMI interrupt is signaled on a non-I/O instruction boundary, setting the I/O instruction restart field to FFH prior to executing the RSM instruction will likely result in a program error.

13.12.1. Back-to-Back SMI Interrupts When I/O Instruction Restart Is Being Used

If an SMI interrupt is signaled while the processor is servicing an SMI interrupt that occurred on an I/O instruction boundary, the processor will service the new SMI request before restarting the originally interrupted I/O instruction. If the I/O instruction restart field is set to FFH prior to returning from the second SMI handler, the EIP will point to an address different from the originally interrupted I/O instruction, which will likely lead to a program error. To avoid this situation, the SMI handler must be able to recognize the occurrence of back-to-back SMI interrupts when I/O instruction restart is being used and insure that the handler sets the I/O instruction restart field to 00H prior to returning from the second invocation of the SMI handler.

13.13. SMM MULTIPLE-PROCESSOR CONSIDERATIONS

The following should be noted when designing multiple-processor systems:

- Any processor in a multiprocessor system can respond to an SMM.
- Each processor needs its own SMRAM space. This space can be in system memory or in a separate RAM.
- The SMRAMs for different processors can be overlapped in the same memory space. The only stipulation is that each processor needs its own state save area and its own dynamic data storage area. (Also, for the Pentium and Intel486 processors, the SMBASE address must be located on a 32-KByte boundary.) Code and static data can be shared among processors. Overlapping SMRAM spaces can be done more efficiently with the P6 family processors because they do not require that the SMBASE address be on a 32-KByte boundary.
- The SMI handler will need to initialize the SMBASE for each processor.
- Processors can respond to local SMIs through their SMI# pins or to SMIs received through the APIC interface. The APIC interface can distribute SMIs to different processors.

- Two or more processors can be executing in SMM at the same time.
- When operating Pentium processors in dual processing (DP) mode, the SMIACT# pin is driven only by the MRM processor and should be sampled with ADS#. For additional details, see Chapter 14 of the *Pentium Processor Family User's Manual, Volume 1*.

SMM is not re-entrant, because the SMRAM State Save Map is fixed relative to the SMBASE. If there is a need to support two or more processors in SMM mode at the same time then each processor should have dedicated SMRAM spaces. This can be done by using the SMBASE Relocation feature (see Section 13.11., "SMBASE Relocation").

13.14. ENHANCED INTEL SPEEDSTEP® TECHNOLOGY

Enhanced Intel SpeedStep® Technology on the Pentium M processor efficiently manages processor power consumption via performance state transitions. Processor performance states are defined as discrete operating points associated with different frequencies.

Enhanced Intel SpeedStep Technology on the Pentium M processor differs from previous generations of Intel SpeedStep Technology in two basic ways:

- Centralization of the control mechanism and software interface in the processor by using model-specific registers.
- Reduced hardware overhead; this permits more frequent performance state transitions.

Previous generations of the Intel SpeedStep Technology require processors to be a deep sleep state, holding off bus master transfers for the duration of a performance state transition. Performance state transitions under the Enhanced Intel SpeedStep Technology are discrete transitions to a new target frequency.

Support is indicated by CPUID, using ECX feature bit 07. Enhanced Intel SpeedStep Technology is enabled by setting IA32_MISC_ENABLE MSR, bit 16. On reset, bit 16 of IA32_MISC_ENABLE MSR is cleared.

13.14.1. Software Interface For Initiating Performance State Transitions

State transitions are initiated by writing a 16-bit value to the MSR_PERF_CTL register. If a transition is already in progress, transition to a new value will take effect subsequently.

Reads of MSR_PERF_CTL determine the last targeted operating point. The current operating point can be read from MSR_PERF_STATUS. MSR_PERF_STATUS is updated dynamically.

The 16-bit encoding that defines valid operating points is model-specific. Applications and performance tools are not expected to use either MSR_PERF_CTL or MSR_PERF_STATUS and should treat both as reserved. Performance monitoring tools can access model-specific events and report the occurrences of state transitions.

13.15. THERMAL MONITORING AND PROTECTION

The IA-32 architecture provides three mechanisms for monitoring temperature and controlling power consumption of an IA-32 processor:

1. A **catastrophic shutdown detector** that forces processor execution to stop if the processor's core temperature rises above a preset limit.
2. An **automatic thermal monitoring mechanism** that forces the processor to reduce it's power consumption in order to maintain a predetermined temperature limit.
3. A **software controlled clock modulation mechanism** that permits operating system to implement a power management policy to reduce the power consumption of an IA-32 processor; this is in addition to the reduction offered by the automatic thermal monitoring mechanism.

The first mechanism is not visible to software. The other two mechanisms are visible to software using processor feature information returned by executing CPUID with EAX = 1.

The second mechanism, automatic thermal monitoring, provides two modes of operation. One mode modulates the clock duty cycle; the second mode changes the processor's frequency. Both modes are used to control the core temperature of the processor.

The third mechanism modulates the clock duty cycle of the processor. As shown in Figure 13-6, the phrase 'duty cycle' does not refer to the actual duty cycle of the clock signal. Instead it refers to the time period during which the clock signal is allowed to drive the processor chip. By using the stop clock mechanism to control how often the processor is clocked, processor power consumption can be modulated.

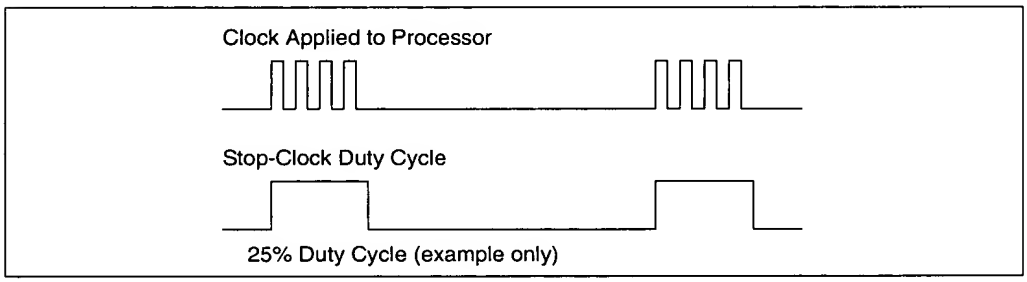

Figure 13-6. Processor Modulation Through Stop-Clock Mechanism

13.15.1. Catastrophic Shutdown Detector

P6 family processors introduced a thermal sensor that acts as a catastrophic shutdown detector. When processor core temperature reaches a factory preset level, the thermal sensor trips and processor execution is halted until after the next reset cycle. This mechanism provides catastrophic over-temperature protection for the processor.

The catastrophic shutdown detector is also implemented in Pentium 4, Intel Xeon and Pentium M processors. It is always enabled.

13.15.2. Thermal Monitor

Pentium 4, Intel Xeon and Pentium M processors include a second temperature sensor that is factory-calibrated to trip when the processor's core temperature crosses a level corresponding to the recommended thermal design envelop. The trip-temperature of the second sensor is calibrated below the temperature assigned to the catastrophic shutdown detector.

13.15.2.1. THERMAL MONITOR 1

The Pentium 4 processor uses the second temperature sensor in conjunction with a mechanism called TM1 (Thermal Monitor 1) to control the core temperature of the processor. TM1 controls the processor's temperature by modulating the duty cycle of the processor clock. Modulation of duty cycles is processor model specific. Note that the processors STPCLK# pin is not used here; the stop-clock circuitry is controlled internally.

Support for TM1 is indicated by CPUID EDX feature bit 29.

TM1 is enabled by setting the thermal-monitor enable flag (bit 3) in IA32_MISC_ENABLE [see Appendix B, *Model-Specific Registers (MSRs)*]. Following a power-up or reset, the flag is cleared, disabling TM1. The basic input/output system (BIOS) is required to enable only one automatic thermal monitoring modes. Operating systems and applications must not disable the operation of these mechanisms.

13.15.2.2. THERMAL MONITOR 2

The Intel Pentium M processor provides an additional automatic mechanism called TM2 (Thermal Monitor 2) to control the core temperature of the processor. TM2 controls temperature by reducing the operating frequency and voltage of the processor. TM2 offers a higher performance level for a given level of power reduction than TM1.

Note that TM2 is triggered by the same temperature sensor in the processor as TM1. Support for TM2 is indicated by CPUID ECX feature bit 8.

NOTE

The mechanism to enable TM2 may be implemented differently in future IA-32 processors.

On Pentium M processors, TM2 is enabled if the TM_SELECT flag (bit 16) of the MSR_THERM2_CTL register is set to 1 and bit 3 of the IA32_MISC_ENABLE register is set to 1.

Following a power-up or reset, the TM_SELECT flag is cleared. BIOS is required to enable either TM1 or TM2. Operating systems and applications must not disable the mechanisms that enable TM1or TM2. On Pentium M processors, if bit 3 of the IA32_MISC_ENABLE register is set and TM_SELECT flag of the MSR_THERM2_CTL register is cleared, TM1 is enabled.

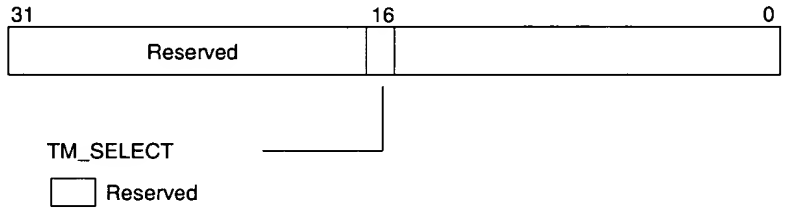

Figure 13-7. MSR_THERM2_CTL Register

13.15.2.3. PERFORMANCE STATE TRANSITIONS AND THERMAL MONITORING

If the thermal control circuitry (TCC) for thermal monitor (TM1/TM2) is active, writes to the MSR_PERF_CTL will effect a new target operating point specified in the MSR_PERF_CTL register.

If TM1 is enabled and the TCC is engaged, the performance state transition can commence before the TCC is disengaged. If TM2 is enabled and the TCC is engaged, the performance state transition specified by a write to the MSR_PERF_CTL will commence after the TCC has disengaged.

13.15.2.4. THERMAL STATUS INFORMATION

The status of the temperature sensor that triggers the thermal monitor (TM1/TM2) is indicated through the thermal status flag (bit 0) and thermal status log flag (bit 1) in the IA32_THERM_STATUS MSR (see Figure 13-8).

The functions of these flags are:

Thermal Status flag, bit 0
When set, indicates that the processor core temperature is currently at the trip temperature of the thermal monitor and that the processor power consumption is being reduced via either TM1 or TM2, depending on which is enabled. When clear, the flag indicates that the core temperature is below the thermal monitor trip temperature. This flag is read only.

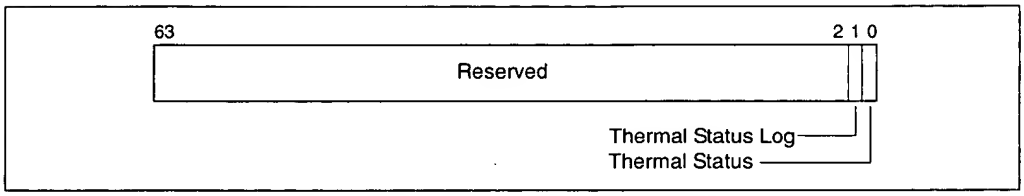

Figure 13-8. IA32_THERM_STATUS MSR

Thermal Status Log flag, bit 1
When set, indicates that the thermal sensor has tripped since the last power-up or reset or since the last time that software cleared this flag. This flag is a sticky bit; once set it remains set until cleared by software or until a power-up or reset of the processor. The default state is clear.

After the second temperature sensor has been tripped, the thermal monitor (TM1/TM2) will remain engaged for at least 1 ms or until the processor core temperature drops below the preset trip temperature of the temperature sensor, taking hysteresis into account.

While the processor is in a stop-clock state, interrupts will be blocked from interrupting the processor. This holding off of interrupts increases the interrupt latency, but does not cause interrupts to be lost. Outstanding interrupts remain pending until clock modulation is complete.

The thermal monitor can be programmed to generate an interrupt to the processor when the thermal sensor is tripped. The delivery mode, mask and vector for this interrupt can be programmed through the thermal entry in the local APIC's LVT (see Section 8.5.1., "Local Vector Table"). The low-temperature interrupt enable and high-temperature interrupt enable flags (bits 0 and 1, respectively) in the IA32_THERM_INTERRUPT MSR (see Figure 13-9) control when the interrupt is generated; that is, on a transition from a temperature below the trip point to above and/or vice-versa.

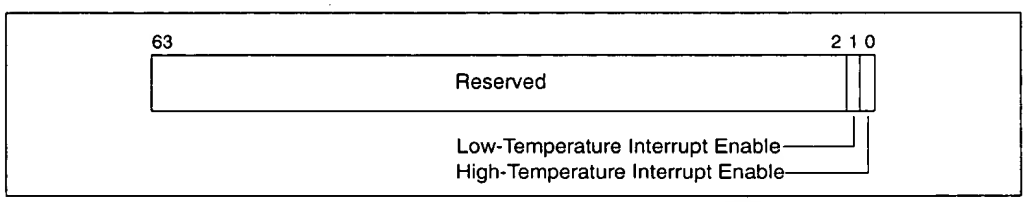

Figure 13-9. IA32_THERM_INTERRUPT MSR

Low-Temperature Interrupt Enable flag, bit 1
Enables an interrupt to be generated on the transition from a high-temperature to a low-temperature when set; disables the interrupt when clear.

High-Temperature Interrupt Enable flag, bit 0
Enables an interrupt to be generated on the transition from a low-temperature to a high-temperature when set; disables the interrupt when clear.(R/W).

The thermal monitor interrupt can be masked by the thermal LVT entry. After a power-up or reset, the low-temperature interrupt enable and high-temperature interrupt enable flags in the IA32_THERM_INTERRUPT MSR are cleared (interrupts are disabled) and the thermal LVT entry is set to mask interrupts. This interrupt should be handled either by the operating system or system management mode (SMM) code.

Note that the operation of the thermal monitoring mechanism has no effect upon the clock rate of the processor's internal high-resolution timer (time stamp counter).

13.15.3. Software Controlled Clock Modulation

Pentium 4, Intel Xeon and Pentium M processors also support software-controlled clock modulation. This provides a means for operating systems to implement a power management policy to reduce the power consumption of the processor. Here, the stop-clock duty cycle is controlled by software through the IA32_THERM_CONTROL MSR (see Figure 13-10).

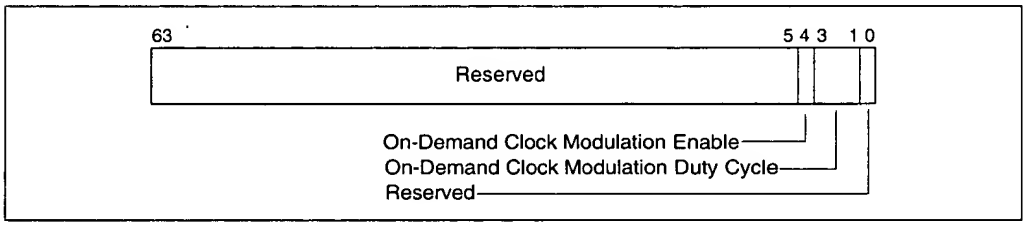

Figure 13-10. IA32_THERM_CONTROL MSR

The IA32_THERM_CONTROL MSR contains the following flag and field used to enable software-controlled clock modulation and to select the clock modulation duty cycle.

On-Demand Clock Modulation Enable, bit 4
Enables on-demand software controlled clock modulation when set; disables software-controlled clock modulation when clear.

On-Demand Clock Modulation Duty Cycle, bits 1 through 3
Selects the on-demand clock modulation duty cycle (see Table 13-5). This field is only active when the on-demand clock modulation enable flag is set.

Note that the on-demand clock modulation mechanism (like the thermal monitor) controls the processor's stop-clock circuitry internally to modulate the clock signal. The STPCLK# pin is not used.

Table 13-5. On-Demand Clock Modulation Duty Cycle Field Encoding

Duty Cycle Field Encoding	Duty Cycle
000B	Reserved
001B	12.5% (Default)
010B	25.0%
011B	37.5%
100B	50.0%
101B	63.5%
110B	75%
111B	87.5%

The on-demand clock modulation mechanism can be used to control processor power consumption. Power management software can write to the IA32_THERM_CONTROL MSR to enable clock modulation and to select a modulation duty cycle. If on-demand clock modulation and TM1 are both enabled and the thermal status of the processor is hot (bit 0 of the IA32_THERM_STATUS MSR is set), clock modulation at the duty cycle specified by TM1 takes precedence, regardless of the setting of the on-demand clock modulation duty cycle.

For Hyper-Threading Technology enabled processors, the IA32_THERM_CONTROL register is duplicated for each logical processor. In order for the On-demand clock modulation feature to work properly, the IA32_THERM_CONTROL register must be programmed identically on all logical processors in the same physical processor.

For the P6 family processors, on-demand clock modulation was implemented through the chipset, which controlled clock modulation through the processor's STPCLK# pin.

13.15.4. Detection of Thermal Monitor and Software Controlled Clock Modulation Facilities

The ACPI flag (bit 22) of the CPUID feature flags indicates the presence of the IA32_THERM_STATUS, IA32_THERM_INTERRUPT, and IA32_THERM_CONTROL MSRs, and the xAPIC thermal LVT entry.

The TM1 flag (bit 29) of the CPUID feature flags indicates the presence of the automatic thermal monitoring facilities that modulate clock duty cycles.